Poster: Compiler-assisted Automatic Checkpointing for Transiently-powered Embedded Devices

Naveed Anwar Bhatti
Politecnico di Milano, Italy
naveedanwar.bhatti@polimi.it

Luca Mottola
Politecnico di Milano, Italy and SICS Swedish ICT
luca.mottola@polimi.it

Abstract

We present compiler-assisted automatic checkpoint strategies to allow transiently-powered embedded sensing devices efficiently save the system's state onto non-volatile memory before energy is exhausted. Our solution operates at compile-time with no developer intervention based on the control-flow graph (CFG) of a program, while adapting to varying levels of remaining energy and all possible program executions at run-time. In addition, the underlying design rationale allows the system to spare the energy-intensive probing of the energy buffer whenever possible.

1 Introduction

Advances in energy harvesting and wireless energy transfer are redefining the scope and extent of the energy constraints in embedded sensing [4]. However, energy provisioning from ambient harvesting or wireless transfer is generally erratic; therefore, devices need to cope with highly variable, yet unpredictable energy supplies, and be prepared to survive periods of energy unavailability.

One way to enable the operation of such transiently-powered devices is to efficiently checkpoint the system's state on non-volatile memory [2, 7] whenever energy is about to be exhaust too early. However, the more crucial aspect is to know *when* and *how* to perform the checkpoint. Doing it early would essentially correspond to a waste of energy that could be usefully employed in further computations. In contrast, excessively postponing a checkpoint may yield a situation where insufficient energy is left to complete the operation. Because of the unpredictable supply of energy and the varying run-time execution of programs, striking an efficient trade-off is challenging.

We present compiler-assisted automatic checkpoint strategies to place calls to *trigger* functions. *Trigger* functions are responsible for triggering of checkpointing mechanism with-in the user code after checking some conditions. Our solution looks at the control-flow graph (CFG) of a program and places triggers according to different strategies depending on the programming constructs. Based on available energy as well as the worst-case estimation of the energy required to reach the next trigger call, our solution decides whether to perform the checkpoint before proceeding with the execution. Also, it can dynamically adapt to varying levels of remaining energy at run-time and allows the system to spare energy-intensive probing of the energy buffer, for example, through ADCs, whenever possible.

2 Approach

Calls to trigger functions placed in the code essentially represent two types of overhead compared to the normal computation. First overhead is the energy cost of checking whether a checkpoint is necessary or not, which is normally constant. Second overhead is the energy cost of the actual checkpoint. This overhead depends on where in the program the checkpoint occurs, making its energy cost typically proportional to the size of the allocated memory when the checkpoint takes place [2].

Our goal is to minimize these two overheads by *i)* minimizing the number of trigger calls that are uselessly executed, *ii)* postponing the actual checkpoint to when the available energy is strictly sufficient to that end, and *iii)* minimizing checkpoint size. The first two needs are at odds with each other. Postponing the checkpoint, in fact, requires to frequently check how close is the execution to when no sufficient energy is left to perform the checkpoint. However, the energy cost of frequent trigger calls, especially if every such call needs to probe the energy buffer through ADC calls, may become prohibitive.

To handle this issue, we give the ability to trigger functions to reason on whether the system can reach the *next* trigger call or not. As a result, every trigger call can take an informed decision on whether to checkpoint. Say E_{next} is the energy to execute the required MCU cycles from the T_{i-1}-th call to the T_i-th call, whereas $E_{CKP(i)}$ is the energy required to checkpoint the system's state against the size of the allocated memory at the T_i-th call, as intuitively depicted in Figure 1. A checkpoint at the T_{i-1}-th call is required if

$$E_{remaining} \leq E_{next} + E_{CKP(i)} \quad (1)$$

where $E_{remaining}$ is the energy left in the buffer when executing the T_{i-1}-th trigger call.

International Conference on Embedded Wireless Systems and Networks (EWSN) 2017
20–22 February, Uppsala, Sweden

ISBN: 978-0-9949886-1-4

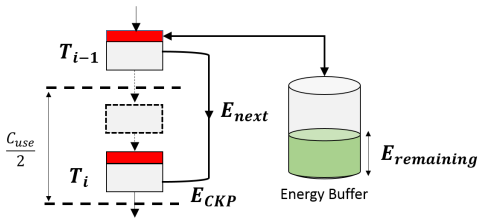
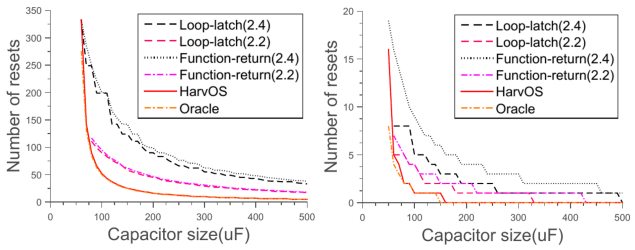


Figure 1: Decision logic to take a checkpoint.



(a) Kalman filter. (b) AES.

Figure 2: Total resets necessary to complete a workload.

At run-time, we can obtain the value of $E_{remaining}$ through software-based techniques [5, 1] or dedicated hardware solutions [6] with negligible overhead. The ability to reason on whether the system can reach the *next* trigger call is one of the key of traits of our approach, and a major source of improvements compared to previous works.

To optimize the placement of trigger calls, we rely on the CFGs of the program and compile-time information on memory allocation patterns. Static code analysis techniques exist that can return accurate information on the evolution of the stack and, in many cases, of the heap as well [3]. We split the CFG into sub-graphs and within each sub-graph, we identify the block corresponding to the minimum size of allocated memory, and place a trigger call right at the end of it. This reduces the cost of checkpointing, as the amount of data to copy over stable storage is minimal within a sub-graph.

Normally, CFGs show complex structures reflecting the variety of available programming constructs, such as branching statements, loops, and function calls. This means there may be multiple places in a sub-graph corresponding to the minimum allocated memory, as a function of different execution paths. Moreover, embedded devices often operate in an interrupt-driven manner, that is, the execution through a CFG may be arbitrarily preempted and temporarily redirected through the CFG of interrupt handlers. We develop a set of trigger placement rules that, depending on the program structure, dictate where to place the trigger call and what to consider as the E_{next} energy to reach the next trigger call. The complete set of rules is *recursively* applied until an elementary block in the CFG is reached. The rules also determine the conditions when probing the energy buffer for the value of $E_{remaining}$ is strictly needed, or $E_{remaining}$ can be inferred from compile-time information. The latter situation allows the system to spare operations that may be energy-expensive per se, such as probing ADCs.

3 Preliminary Evaluation

Our evaluation considers modern 32-bit MCUs, such as the ARM Cortex M3 MCU aboard an ST Nucleo L152RE board, and three increasingly complex benchmark codes commonly employed in embedded applications: Kalman filter, finite impulse response (FIR) filter, and Advanced Encryption Standard (AES). We sweep the possible executions of programs against varying size of the underlying energy buffers to measure the performance of our solution in terms of a number of resets. This figure is inversely proportional to the effectiveness of a given instrumentation strategy. Given a fixed workload, the more the MCU needs to reboot, the more the checkpointing operation is subtracting energy from useful computations. Lower values are thus better.

Baselines. We consider the *loop-latch* and *function-return* strategies of MementOS. The former places trigger calls at the end of loop iterations; the latter places trigger calls at function return points. In addition, we apply a brute-force search on all possible executions of the code to identify an *oracle* that, by predicting how the execution is going to unfold in the future, knows the last practical point in time when to checkpoint. This is not feasible in reality; to make it work in a concrete execution, one would theoretically need to place a trigger call after every instruction in the code, yielding an unbearable overhead.

Results. Figure 2 plots the results we obtain in the number of MCU resets for a fixed workload. As expected, bigger capacitor sizes generally correspond to fewer resets, in that the individual executions can progress farther on a single charge. Compared to either of the MementOS strategies, our solution completes the fixed workload with 69% fewer resets on average, with a peak improvement of 80% fewer resets. Moreover, in most cases, the performance of our solution rests very close to the *oracle*. Notably, for the Kalman filter code, the performance is often almost the same, as the *oracle*. This demonstrates that the rationale explained in Section 2 strikes an effective trade-off between opposite needs, ultimately performing similarly to an optimal solution that is, however, unfeasible in practice. Overall, if we compare Figure 2a, obtained using the Kalman filter code, with Figure 2b that depicts the performance of the AES implementation, one may note that the performance of our solution is robust against diverse benchmark codes with different complexity and structure, unlike existing approaches.

4 References

[1] Bernhard Buchli et al. Battery state-of-charge approximation for energy harvesting embedded systems. In *EWSN*, 2013.

[2] N. A. Bhatti and L. Mottola. Efficient state retention for transiently-powered embedded sensing. In *EWSN*, 2016.

[3] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, 2006.

[4] Naveed Anwar Bhatti et al. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Transactions on Sensor Networks*, 12, 2016.

[5] Philipp Sommer et al. Information bang for the energy buck: Towards energy-and mobility-aware tracking. In *EWSN*, 2016.

[6] Prabal Dutta et al. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN*, 2008.

[7] B. Ransford, J. Sorber, and K. Fu. MementOS: System support for long-running computation on RFID-scale devices. In *ASPLOS*, 2011.